BeatBlendr

Database Design

Comment Received for Stage 1:

What are related applications and how does yours set itself apart? Your project is very ambitious, but it seems to be largely focused on ML and the social aspect. For your project, we ask you to instead focus on the backend development aspect. What interesting queries can you provide and what insights can they provide about the data? Implementing full ML models and social media aspects is not the focus of this course and may be too ambitious. However, for the extra credit creative component, it would be interesting to implement recommendations using ML models.

Actions on the comment

Based on feedback on our previous stage (we were told that our project is ambitious and based on machine learning. We have reduced some features and are implementing recommendations purely on the basis of SQL queries and are not using any machine learning

Recap

Recap of our features which will be implemented using the database schema defined below.

1. Users on our platform can play multiple songs, we will store all user song history and use it for recommendation

2. Based on the genre of the previously played songs, we will filter songs. We will calculate cosine similarity between previous song and filtered songs, order by cosine similarity in ascending order then order by popularity in descending order to decide our order of song recommendation. We will generate recommendations once a day and store them in recommendations table

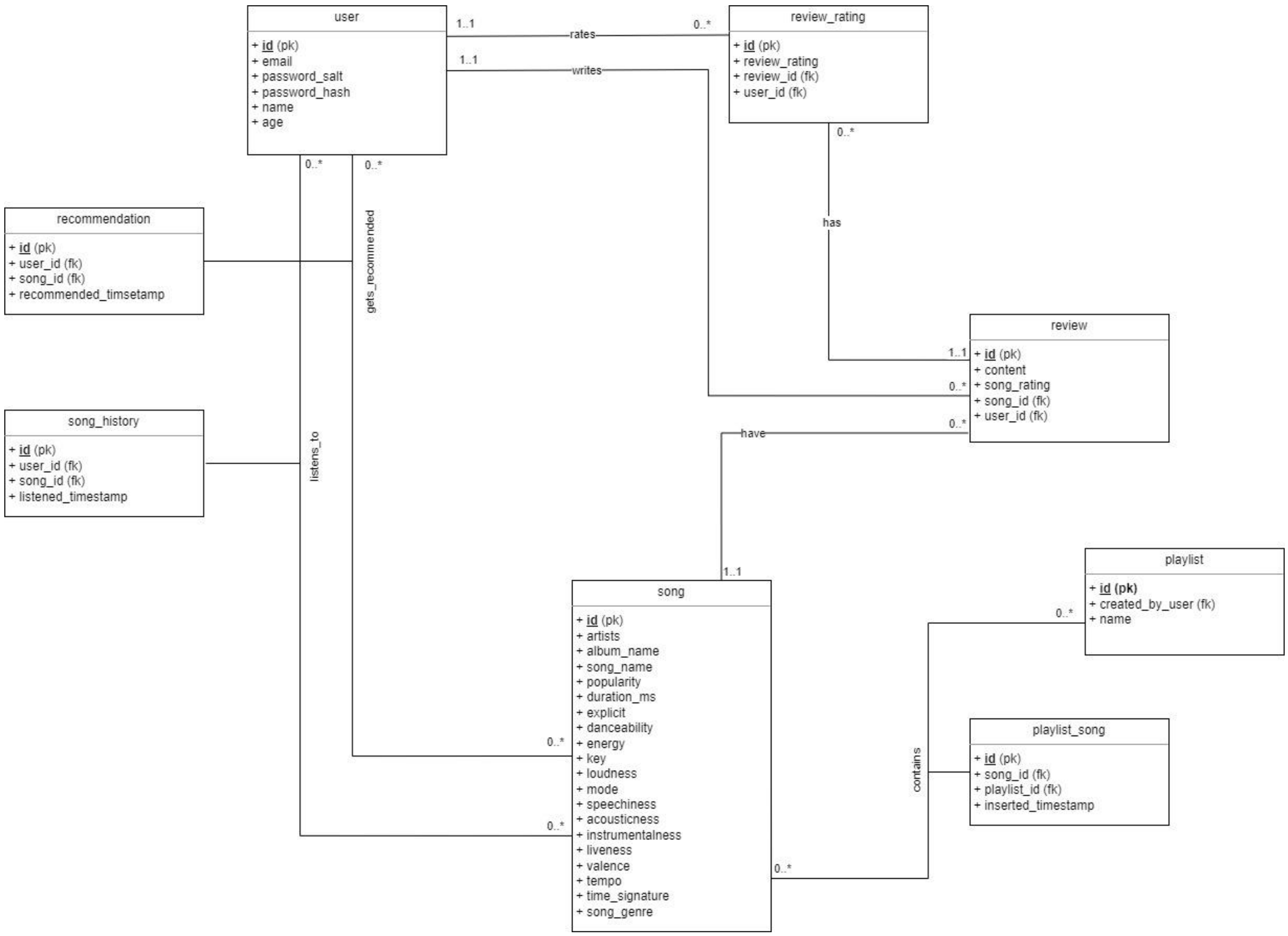
3. User can write reviews for songs

4. These reviews can be rated by other users

5. We calculate the popularity of songs as a weighted average of user review weighted by the rating of that review

6. User can create a playlist and add songs to that playlist

UML Diagram for BeatBlender



Link: [UML diagram on draw.io]()

Assumptions

Relation assumptions and cardinality:

User-review_rating (Rates):

1. One to many
2. User can rate multiple reviews
3. Each review rating will be associated with one user only

User-review relation (Writes):

1. One to many
2. User can write multiple reviews
3. Each review will be associated with one user only

User-song relation (Gets_recommended):

1. Many to many
2. User can gets recommended multiple songs
3. A song can be recommended to multiple users

User-song relation (Listens_to):

1. Many to many
2. User can listen multiple songs
3. A song can be played by multiple users

Review-review rating relation (has):

1. One to many
2. Review can have multiple review_ratings
3. Each review rating will be associated with only 1 review

Songs-review relations (have):

1. One to many
2. Song can have multiple reviews
3. Each review will be associated with 1 song only

Song-playlist (contains):

1. Many to many
2. Song can be a part of multiple playlists
3. A playlists can be multiple songs

Entity Assumptions

** All Identifier (ID) fields are stored using the base62 encoding(https://en.wikipedia.org/wiki/Base62) which is widely used by Meta, Instagram, Spotify, etc. It uses 26(lowercase) + 26(uppercase) + 10(digits) = 62 characters for encoding.

user

1. User authentication information will not be stored in plain text. Instead, the password value is salted and hashed and then stored, salt will be unique for each user
2. Password hash and salt have 32 bytes because we are using the SHA256 algorithm for storing these values

song

1. The song entity contains descriptive metadata values
2. The magnitude of each decimal attribute inidicates its strength and integer values are used for categorical data

song_history

1. The song history entity stores data about users' listening history
2. This entity will be used for generating song recommendations for the user

recommendation

1. The recommendation entity stores song recommendations for a user based on their listening history
2. Recommendations are timestamped to identify their relevance and later discard them

playlist_song

1. The playlist_song entity stores information about songs that are part of a particular playlist
2. The playlist is created by a user and can be shared with other users for listening, but cannot be edited

playlist

1. The playlist entity stores metadata associated with a playlist and the user who created this entity

review

1. The review entity stores data about a review that is written for a song by a user

review rating

1. Ratings given by users for a song review written by some other user are stored in this entity

Functional Dependencies

FDs for song

id + = {artist, album_name, song_name, popularity, duration_ms, explicit, danceability, energy, key, loudness, mode, speechiness, acousticness, instrumentalness, liveness, valence, tempo,time_signature, song_genre}

id -> artist

id -> album_name

id -> song_name

id -> popularity

id -> duration_ms

id -> explicit

id -> danceability

id -> energy

id -> key

id -> loudness

id -> mode

id -> speechiness

id -> acousticness

id -> instrumentalness

id -> liveness

id -> valence

id -> tempo

id -> time_signature

id -> song_genre

FDs for song_history

id+ = {user_id, song_id, listened_timestamp}

id -> user_id

id -> song_id

id -> listened_timestamp

FDs for recommendation

id+ = {user_id, song_id, recommended_timsetamp}

id -> user_id

id -> song_id

id -> recommended_timsetamp

FDs for playlist_song

id+ = {song_id, playlist_id, inserted_timestamp}

id -> song_id

id -> playlist_id

id -> inserted_timestamp

FDs for playlist

id+ = {created_by_user, name}

id -> created_by_user

id -> name

FDs for review

id+ = {id, content, song_rating, song_id, user_id, content, song_rating}

id -> content

id -> song_rating

id -> song_id

id -> user_id

user_id, song_id -> content

user_id, song_id -> song_rating

user_id, song_id -> id

FDs for review_rating

id+ = {id, review_rating, review_id, user_id, review_rating}

id -> review_rating

id -> review_id

id -> user_id

review_id, user_id -> review_rating

review_id, user_id -> id

FDs for user

id+ = {id, email, name, age, password_salt, password_hash}

id -> email

id -> name

id -> age

id -> password_salt

id -> password_hash

All the relations above have their corresponding id column as their super key and as the functional dependencies show that the id column determines all the other attributes, hence all these relations are in ***BCNF.***

Why BCNF over 3 NF?

BCNF has low redundancy compared to 3 NF. We are not losing any important functional dependencies while normalizing our schema to BCNF. We chose to use BCNF for these reasons.

Relational Schema

song(id: varchar(22) [PK],
artist: varchar(100),
album_name: varchar(100),
song_name: varchar(100),
popularity: double,
duration_ms: int,
explicit: boolean,
danceability: decimal,
energy: decimal,
key: int,
loudness: decimal,
mode: int,
speechiness: decimal,
acousticness: decimal,
instrumentalness: decimal,
liveness: decimal,
valence: edcimal,
tempo: decimal,
time_signature: int,
song_genre: varchar(50))

song_history(id: int [PK]
user_id: varchar(22) [FK to user.id]
song_id: varchar(22) [FK to song.id],
listened_timestamp: timestamp)

recommendation(id: int [PK],
user_id (fk): varchar(22),
song_id (fk): int,
recommended_timsetamp: timestamp)

playlist_song (id: varchar(22) [PK],
song_id: varchar(22) [FK to song.id],
playlist_id: varchar(22) [FK to playlist.id],
inserted_timestamp: timestamp)

playlist(id: varchar(22) [PK],
created_by_user: varchar(22) [FK to user.id],
name: varchar(100))

review(id: varchar(22) [PK],
content: varchar(1000),
song_rating: int,
song_id: varchar(22) [FK to song.id],
user_id: varchar(22) [FK to user.id])

review_rating(id: varchar(22) [PK],
review_rating: int,
review_id: varchar(22) [FK to review.id],
user_id: varchar(22) [FK to user.id])

user(id: varchar(22) [PK],
email: varchar(150),
name: varchar(50),
age: int,
password_salt: varchar(32),
password_hash: varchar(32))